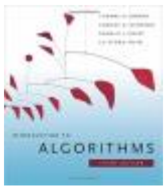

Denison University

Final Semester
Portfolio

CS271 – Spring 2017
Algorithms & Data Structure

James Le

Instructed by
Dr. Jessen Havill

May 8, 2017

Contents

1 Analysis of algorithms

1.1 Preconditions and postconditions

Preconditions and post conditions are comment-like text that are written before each function to illustrate how the function behaves. The precondition shows what the conditions of the program are before entering that function. The postcondition shows what changes that function did to the program.

An example would be:

```
/*=============================================
MinHeap Default Constructor
Precondition: None
Postcondition: Construct a new MinHeap data type
with default size set to n.
=============================================*/
```

In this example, these are the pre-and post-conditions of a constructor. The constructor here has no preconditions because there is no constructed data type. On the other hand, the postcondition explains how the program changed by stating what have occurred in the program after that function.

1.2 Loop invariants

A loop invariant is a condition that is necessarily true immediately before and immediately after each iteration of a loop. The loop invariant has three different parts. The first part is called initialization, which states the initial input variables that are inputted into inside the while loop. The second part is called maintenance, where you prove that the loop stays the same while maintaining the initialized property. The final part is called termination. Termination is the condition when the loop ends and also when the loop invariants end as well.

1.3 Analyzing the asymptotic time complexity of iterative and recursive algorithms

Recursive

Recursive algorithms are functions that call themselves again and again until the base condition is satisfied. A recursive function is a function with the format like a normal equation starting with T(n). This format is called a recurrence relation between the recursive function and the entire body of the code. A recurrence relation is also known as a difference equation.

Recursive algorithms are mostly used to solve complicated problems when their application is easy and effective.

Iterative

An iterative algorithm will use looping statements such as for loop, while loop or do-while loop to repeat the same steps. An iterative algorithm will be faster than the recursive algorithm because of overheads like calling functions and registering stacks repeatedly. For example, the following code:

```
LinearSearch(int A [])
{
   int v = A[0];
   for(int  i = 1; i <= A.length; i++)
   {
         if(v == A[i])
         {
            return i;
         }
   }
   return NIL
}
```

The worst-case scenario here happens when the item is found at the very end of the array. This means that the entire function has the value of the iterative time complexity, which is n, plus a constant number of steps. So when we take the Θ, it will be Θ(n), apparent from this function. Of course, each iterative function has a different time complexity depending on the number of iterations per function.

2 Linear data structures

2.1 Proper usage and design of List, Queue, and Stack ADTs

List

A list is an ADT that represents a countable number of ordered values, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a finite sequence; the potentially infinite analog of a list is a stream. Lists are a basic example of containers, as they contain other values. If the same value occurs multiple times, each occurrence is considered a distinct item. The visualization of a list is shown below.

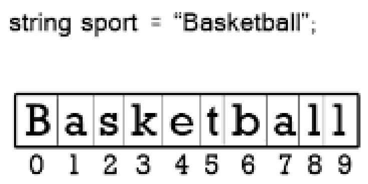

Figure 2: A Simple List

Each letter of the string is stored in its own slot. Each letter is accessible through a specific index. For example, the letter "B" is accessible through two different indices, one for the "B" at the beginning of the word and one for the" b " in the middle of the word. Each letter can be accessed by using a for loop to iterate through the list.

Queue

Queue is an ADT, in which the first element is inserted from one end called the tail, and deletion of existing element takes place from the other end called head. This makes queue as FIFO data structure, which means that element inserted first will also be removed first. The process to add an element into queue is called **Enqueue** and the process of removal an element from queue is called **Dequeue.**

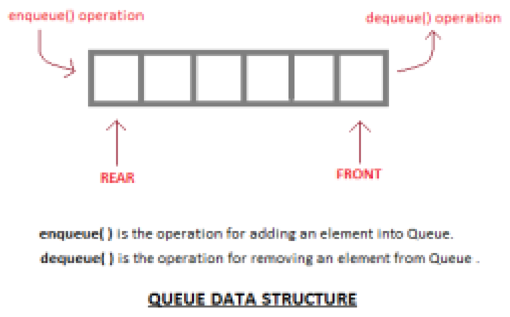
enqueue() operation
dequeue() operation
REAR
FRONT
enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue .
QUEUE DATA STRUCTURE

Figure 3: A Simple queue

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1 – Serving requests on a single shared resource, like a printer, CPU task scheduling.

2 – In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

3 – Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive (first come, first served).

Stack

Stack is an ADT with a bounded (predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects. Stack is a LIFO (Last In, First Out) structure.

The simplest application of a stack is to reverse a word. You push a given word to stack – letter by letter – and then pop letters from the stack. There are other uses also like: Parsing, Expression Conversion (Infix to Postfix, Postfix to Prefix etc) and many more.

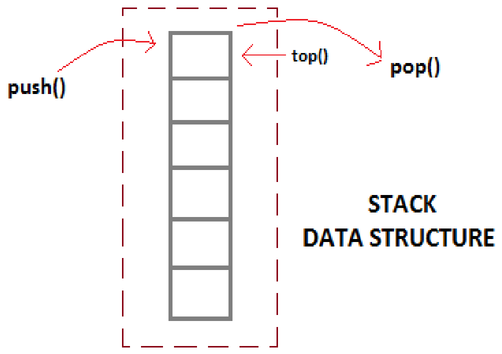

Figure 4: A Simple Stack

2.2 Implementation using both dynamically allocated arrays and linked list data structures

Array

An array is a container which can hold a fix number of items and these items should be of the same type. Each item stored in array is called an **element**, each location of an element in an array has a numerical **index**, which is used to identify the element.

Linked List

A linked list is a linear data structure where each element is a separate object. Each element of a list is comprising of 2 items – the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the head of the list.

The difference between a linked list and an array is that a linked list is easier to access and parse through as it has pointers. Pointers could be pointed to any of the values inside. This was proven in the implementation of hash table, which is one of our project during the semester. The hash table was an adjacency list that contained to two linked list. Through the characteristics of our linked list, the hash table had an amazing run time due to the structure of the linked list. Ideally in the best-case scenario, the adjacency list would be a list pointing to another list that contains an element each depending on our hash function. The functions of the list such as get, remove, insert or append, made it much easier to access the hash table than any other data structure we used.

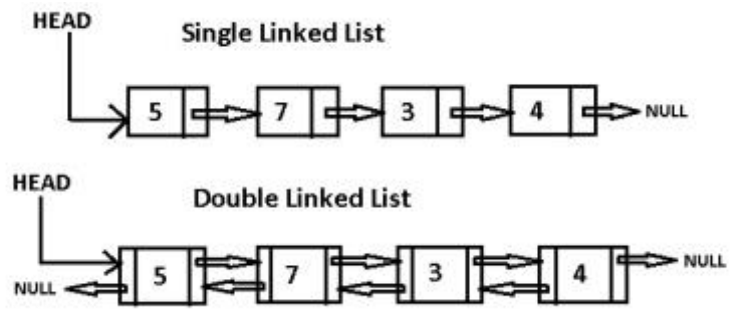

Figure 5: Linked Lists Examples

Singly linked lists contain nodes which have a data part as well as an address part, i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal. In a doubly linked list, each node contains two links: the 1st link points to the previous node and the next link points to the next node in the sequence.

Array or Linked List

Here is the table of the running times of 3 main methods used in both arrays and linked list:

	Unsorted Array	Sorted Array	Linked List
get	θ(n)	θ(log(n))	θ(n)
insert	θ(1)	θ(n)	θ(1)
remove	θ(n)	θ(log(n))	θ(n)

Linked Lists are preferred over arrays when: (a) you need constant-time insertions/deletions from the list, (b) you don’t know how many items will be in the list, (c) you don’t need random access to any elements, and (d) you want to be able to insert items in the middle of the list (such as a priority queue).

Arrays are preferred over linked lists when: (a) you need indexed/random access to elements, (b) you know the number of elements in the array ahead of time so that you can allocate the correct amount of memory for the array, (c) you need speed when iterating through all the elements in sequence, and (d) memory is a concern. Filled arrays take up less memory than linked lists. Each element in the array is just the data. Each linked list node requires the data as well as one pointers to the other elements in the linked list.

3 Priority queues and binary heaps

3.1 Usage and Design of a priority queue ADT and implementation with heap

Design and Usage

In computer science, a priority queue is an ADT which is like a regular queue data structure, but where additionally each element has a **priority** associated with it. Priority queue can be used in many situations:

- Event-driven simulation: customers in a line, colliding particles
- Numerical computation: reducing round-off error
- Data compression: Huffman codes which compresses data
- Graph searching: Dijkstra's algorithm, Prim's algorithm
- Number theory: sum of powers
- Artificial Intelligence: A* search
- Statistics: maintain largest M values in a sequence
- Operating systems: load balancing, interrupt handling
- Discrete optimization: bin packing, scheduling
- Spam filtering: Bayesian spam filter

Implementation

A priority queue can be implemented using many of the data structures that we've already studied. However, those data structures do not provide the most efficient operations. To make all the operations very efficient, we'll use heap. Here is the constructor for the heap that is called on the constructor of the priority queue to make it:

```
Template <class KeyType>
MinHeap<KeyType>::MinHeap(int  n) // Constructor
{
    heapSize = 0;
    capacity = n;
    A = new KeyType [capacity];
}
```

The function creates an array with a specific size. It would later call on heapify, buildHeap, and heapsort functions in order to create the ADT with customized specifications.

4 Graphs

4.1 The two main ways to implement a graph ADT

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**. Formally, a graph is a pair of sets V, E, where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. There are 2 main ways to represent graphs: Adjacency List and Adjacency Matrix.

Adjacency List

Representing a graph with adjacency lists combines adjacency matrices with edge lists. For each vertex I, store an array of the vertices adjacent to it. We typically have an array of |V| adjacency lists, one adjacency list per vertex. Here's an adjacency-list representation of the social network graph:

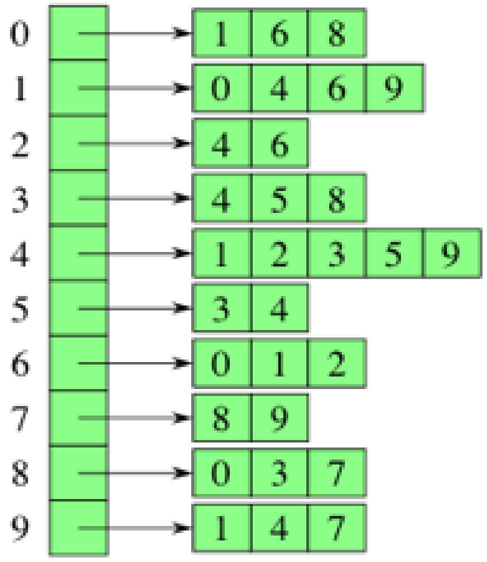

Figure 6: An Adjacency List Using Linked Lists

Adjacency Matrix

For a graph with |V| vertices, an adjacency matrix is a |V| x |V| matrix of 0s and 1s, where the entry in row I and column j is 1 if and only if the edge (i, j) is in the graph. If you want to indicate an edge weight, put it in the row i, column j entry, and reserve a special value (perhaps NULL) to indicate an absent edge. Here' the adjacency-matrix for the social network graph:

$$
\begin{array}{c|cccccccccc|}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
5 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
6 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
8 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
9 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
\hline
\end{array}
$$

Figure 7: An Adjacency Matrix

4.2 DFS and BFS algorithms, implementations and usage

BFS

BFS is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Given a graph G = (V, E) and a distinguished source vertex s, BFS systematically explores the edges of G to “discover” every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a “breadth-first tree” with root s that contains all reachable vertices. For any vertex v reachable from s, the simple path in the breadth-first tree from s to v corresponds to a “shortest-path” from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs. To keep track of progress, BFS colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is discovered the first time it is encountered during the search, at which time it becomes non-white. Gray and black vertices, therefore, have been discovered, but BFS distinguishes between them to ensure that the search proceeds in a breadth-first manner.

```
BFS(G, s)
{
    for each vertex u in G.V - {s}
    {
       u.color = WHITE
       u.d = infinity
       u.parent = NIL
    }
    s.color D GRAY
    s.d = 0
    s.parent = NIL
    Q = NULL
    ENQUEUE(Q.s)
    while Q != NULL
    {
       u = DEQUEUE(Q)
       for each v in G.Adj[u]
       {
          if v.color == WHITE
          {
             v.color = GRAY
```

```
                v.d = u.d + 1
                v.parent = u
                ENQUEUE(Q,v)
            }
        }
        u.color = BLACK
    }
}
```

The total running time of the BFS procedure is θ(V + E), with V being the vertex and E being the edge. Thus, BFS runs in time linear in the size of the adjacency-list representation of G.

DFS

The strategy followed by DFS is to search "deeper" in the graph whenever possible. DFS explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then DFS selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex. As in BFS, DFS colors vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely.

```
DFS(G)
{
 for each vertex u in G.V
 {
   u.color = WHITE
    u.parent = NIL
 }
 time = 0
 for each vertex u in G.V
 {
    if u.color == WHITE
       DFS-VISIT(G.u)
 }
}

DFS-VISIT(G.u)
{
 time = time++
 u.d = time
 u.color = GRAY
 for each v in G.Adj[u]
 {
    if v.color == WHITE
    {
       v.parent = u
       DFS-VISIT(G)
    }
 }
 u.color = BLACK
 time = time++
 u.f = time
}
```

The running time of DFS is Θ(V + E), with V being the vertex and E being the edge.

4.3 Understanding, implementing, and applying algorithms to find shortest paths

Dijkstra

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex s, it grows a tree T that ultimately spans all vertices reachable from S. Vertices are added to T in order to of distance, i.e., first S, then the vertex closes to S, then the next closest, and so on. Following implementation assumes that the graph G is represented by adjacency lists:

```
DIJKSTRA(G,w,s)
{
 INITIALIZE-SINGLE-SOURCE(G.s)
 S = NULL
 Q = G.V
 while Q = NULL
 {
     u = EXTRACT-MIN(Q)
     S = S ∪ {u}
     for each vertex in G.Adj[u]
      RELAX(u,v,w)
  }
}
```

Dijkstra's algorithm runs in O(|E| lg |V|) time, with V being the vertex and E being the edge.

Bellman-Ford

Bellman-Ford algorithm solves the single-source shortest-path problem in the general case in which edges of a given graph can have negative weight as long as G contains no negative cycles. This algorithm, like Dijkstra's algorithm, uses the notion of edge relaxation but does not use with greedy method. Again, it uses u.d as an upper bound on the distance d[u, v] from u to v.

The algorithm progressively decreases an estimate d[v] on the weight of the shortest path from the source vertex s to each vertex v in V until it achieves the actual shortest-path. The algorithm returns Boolean True if the given graph contains no negative cycles that are reachable from source vertex s, otherwise it returns Boolean False.

```
BELLMAN-FORD(G,w,s)
{
 INITIALIZE-SINGLE-SOURCE(G,w,s)
 for i = 1 to |G.V| - 1
 {
    for each edge (u,v) in G.E
       RELAX(u,v,w)
 }
 for each edge (u,v) in G.E
 {
    if v.d > u.d + w(u.v)
       return FALSE
 }
 return TRUE
}
```

The initialization takes O(v) time, the 2 for loops each take O(E) time -> thus, the Bellman-Ford algorithm runs in O(E) time.

4.4 Minimum spanning tress

Prim

Prim's algorithm has the property that the edges in the set A always form a single tree. The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. Each step adds to the tree A a light edge that connects A to an isolated vertex – one on which no edge of A is incident. This rule adds only edges that are safe for A; therefore, when the algorithm terminates, the edges in A form a minimum spanning tree.

In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A. In the pseudocode below, the connected graph G and the root r of the MST to be grown are inputs to the algorithm. During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute. For each vertex v, the attribute v.key is the minimum weight of any edge connecting v to a vertex in the tree; by convention, v.key = infinity if there is no such edge. The attribute v.parent names the parent of v in the tree.

```
MST-PRIM(G,w,r)
{
 for each u in G.V
 {
    u.key = infinity
    u.parent = NIL
 }
 r.key = 0
 Q = G.V
 while Q != NULL ;
 {
    u = EXTRACT-MIN(Q)
    for each v in G.Adj[u]
```

```
      {
         if v in Q and w(u,v) < v.key
         {
            v.parent = u
            v.key = w(u,v)
         }
      }
 }
}
```

The running time of Prim’s algorithm depends on how we implement the min-priority queue Q. For best case scenario, we can implement Q as a binary min-heap. The total running time for Prim’s algorithm then is O(E lg V).

Kruskal

Kruskal’s algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any 2 trees in the forest, an edge (u, v) of least weight. Let C_1 and C_2 denote the 2 trees that are connected by (u, v). Since (u, v) must be a light edge connecting C_1 to some other tree, (u, v) is a safe edge for C_1. Kruskal’s algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge of least possible weight.

Implementation of Kruskal’s algorithm uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u. Thus, we can determine whether 2 vertices u and v belong the same tree by testing whether FIND-SET(u) equals FIND-SET(v). To combine trees, Kruskal’s algorithm calls the UNION procedure.

```
MST-KRUSKAL(G,w)
{
A = NULL
 for each vertex v in G.V
    MAKE-SET(v)
 sort the edges of G.E into ascending order by weight w
 for each edge (u,v) in G.E, taken in ascending order by weight
    if FIND-SET(u,v) != FIND-SET(v)
    {
       A = A ∪ {(u, v)}
       UNION(u,v)
    }
 return A
}
```

The running time of Kruskal’s algorithm for a graph G = (V, E) depends on how we implement the disjoint-set data structure. In best-case scenario, we use the disjoint-set-forest implementation with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known. The total running time of Kruskal’s algorithm then is O(E lg V).

5 Hash tables

5.1 Map ADT implementation

We implemented a dictionary ADT as a template class that inherits from the hash table template class. In computing, a hash table (or hash map) is a data structure which implements an associative array ADT, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. Each slot has a linked list. The word is then added in to that hash table and the word is retrievable as we have the slot number it was put in and we can parse through the linked list.

5.2 Understanding the purpose and implementation of hash functions and their use in approximating universal hashing

Purpose

Hashing can be used for many purposes:
1 – It can be used to compare large amounts of data. You create the hashes for the data, store the hashes and later if you want to compare the data, you just compare the hashes.
2 – Hashes can be used to index data. They can be used in hash tables to point to the correct row. If you want to quickly find a record, you calculate the hash of the data and directly go to the record where the corresponding hash record is pointing to.
3 – They can be used in cryptographic applications like digital signatures.
4 – Hashing can be used to generate seemingly random strings.

Implementation

Here is an example of a hash method implementation for a Movie class using a hashing algorithm I developed from an in-class project:

```
int Movie::hash(int slots) const
{
   if(title.size() == 1)
   {
      char ch;
      ch = title[0];
      int num = ch;
      return num % slots;
   }

   if(title.size() == 2)
   {
      int asciiCode;
      char cha = title[0];
      char cha1 = title[1];
      asciiCode = cha + cha1;
```

```
        int mult = cha;
        return asciiCode * mult % slots;
    }

    if(title.size() > 2)
    {
        unsigned int asciiCode, asciiCode1;
        asciiCode = 1;
        asciiCode1 = 1;
        int n = 77777;
        int numb = 776887;

        for(int i = 0; i < title.size(); i++)
        {
            char c1 = title[i];
            int num1 = c1;
            asciiCode = (asciiCode *  num1 * n) % numb;
        }

        for(int j = 0; j < title.size() - 1; j++)
        {
            char c2 = title[j];
            int num2 = c2;
            asciiCode1 = (asciiCode1 * num2 * n) % numb;
        }
        return (asciiCode * asciiCode1) / (title.size() * title.size() * title.size()) % slots;
    }
}
```

In the hash function, I first check the length of the string in order to customize how the hash function will calculate the slot for a particular string:

1 – In the case that the string is of length 1, the ASCII value of that particular character is modified by the number of slots and the resulting value is the slot number the object is assigned to.

2 – In the second case, the string has a length of 2, and in this case, both of the letters in the string are translated into their respective ASCII codes and added together. The resulting value is multiplied by the ASCII code of the first character in the string and then modified by the number of slots.

3 – In the final case, the string has a length that is greater than 2. I begin by creating separate seeds, of which are large prime numbers. Next, I first iterate through the characters in the string, and update the value of a variable by itself multiplied by the ASCII value of the character being iterated on, multiplied by the first seed, and finally modified by the second seed. This is then done again for the same string, for all characters in the string except the final one. These 2 values are then multiplied together, and divided by the string size cubed, and then modified by the number of slots to return the slot number for a string.

6 Binary search trees

6.1 Implementing a map ADT using a binary search tree

A Binary Search Tree is a tree in which all the nodes follow the 2 properties: (1) The left subtree of a node has a key less than or equal to its parent node's key, and (2) The right subtree of a node has a key greater than its parent node's key. For a previous project, we compared two movies of type string together. The one with the greater value will abide by the rules of the BST, and the same goes for the opposite. An example for a binary search tree can be shown below:

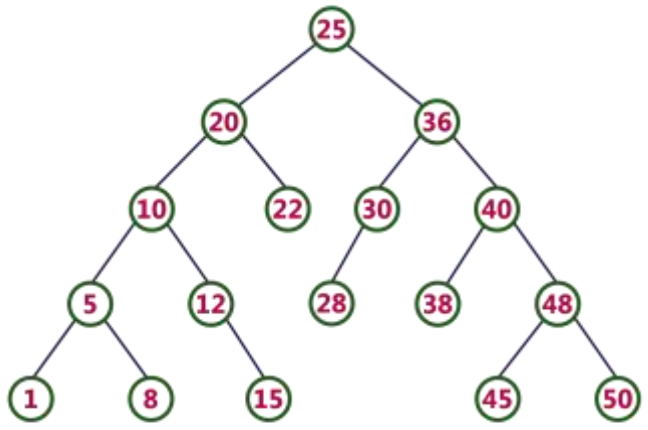

Figure 8: A Binary Search Tree

Here is an example of how an item is inserted and how the decision is made for that item to be in that place:

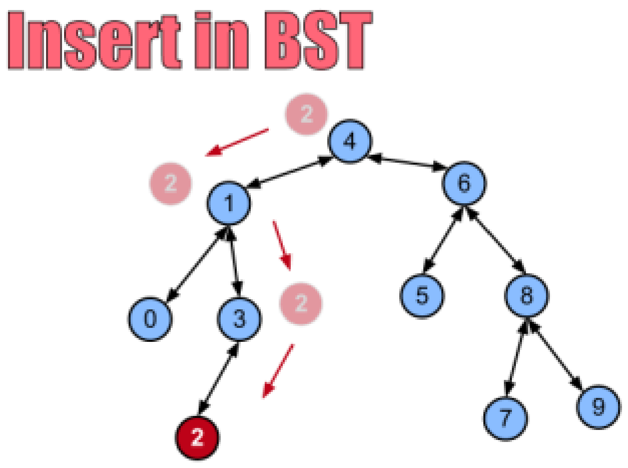

Figure 9: An Insertion into the Binary Search Tree

6.2 Proper usage and implementation of red-black trees

A red-black tree is a kind of self-balancing binary search tree. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions. In addition to the requirements imposed on a binary search tree, the following properties must be satisfied by a red-black tree:

1- The root is black.
2- Each node is either red or black.
3- If a node is red, then both its children are black.
4- A leaves (NIL) are black.
5- Every path from a given node to its descendant NIL leaves contains the same number of black nodes. The number of black nodes from the root to a node is the node's black depth; the uniform number of black nodes in all paths from root to the leaves is called the black-height of the red-black tree.

Here is an example of a red black search tree:

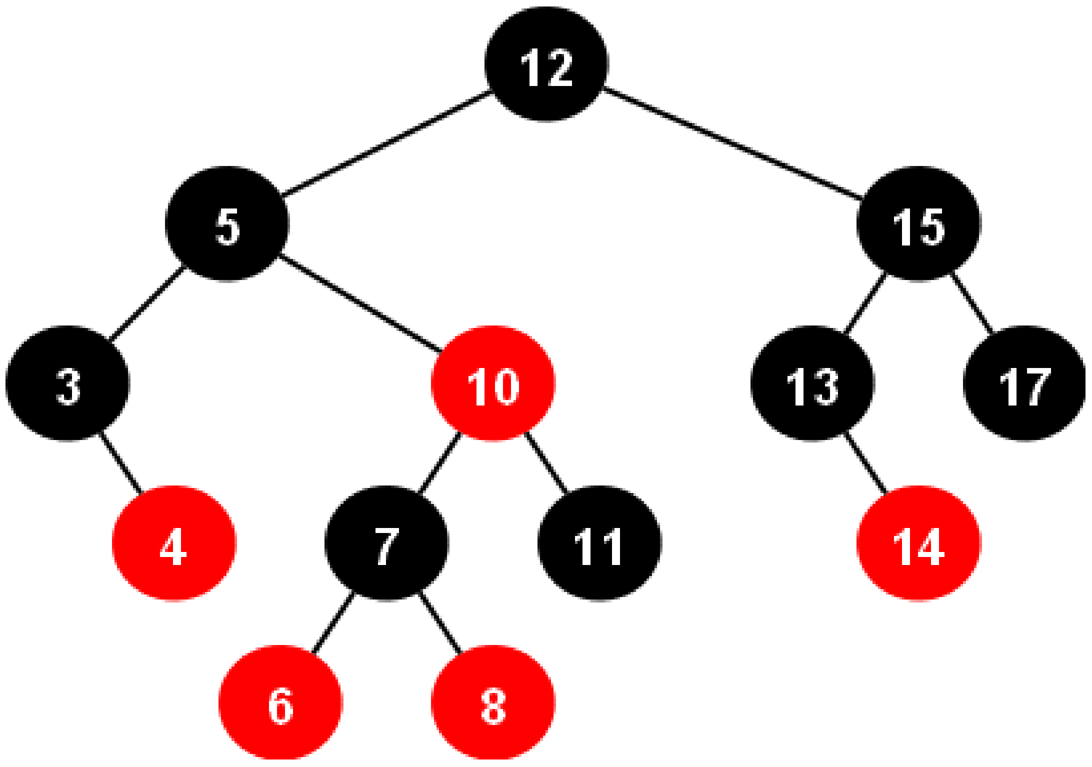

Figure 10: A Red- Black Binary Search Tree

7 Object-oriented programming in C++

7.1 Implementing and solving with template classes in the C++ programming language

The prime purpose of C++ programming was to add objet orientation to the C programming language, which is in itself one of the most powerful programming languages. The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the forms of objects.

Object is the basic unit of OOP, which consist of both data and function that operate on data. When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object. Templates are a way of making your classes more abstract by letting you define the behavior of the class without actually knowing what datatype will be handled by the operations of the class. Below is an example of a templated class:

```
template<class KeyType>
KeyType* MinPriorityQueue<KeyType>::minimum()
{
   if(empty())
   {
      throw EmptyError();
   }
   return A[0];
}
```

7.2 Understanding the concept and application of inheritance and polymorphism

Inheritance

One of the most useful aspects of OOP is code reusability. As the name suggests, inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed as derived class. This is a very important concept of OOP since this feature helps to reduce the code size.

Here is an example of inheritance when defining the class:

```
class Dictionary : public RBT<KeyType>
```

This shows a class called Dictionary inheriting from a red-black tree class.

The next example shows how the structure of the constructor can be inherited:

```
Dictionary() : RBT<KeyType>() { };
```

This shows the Dictionary getting its substructure from the red-black tree class.

The last example shows how the methods are inherited:

```
using RBT<KeyType>::insert;
using RBT<KeyType>::get;
using RBT<KeyType>::toString;
using RBT<KeyType>::empty;
```

This shows what methods from the Red-Black Tree will be used in the Dictionary.

Polymorphism

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism. Below is an example of polymorphism:

```
bool Movies::operator  > (const Movies& mov) const
{
    if(this -> title > mov.title)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

We usually know the > operator to compare numbers. However, after overloading this operator, we have made a new > operator that can compare between two movie title strings in order to give us the one with greater value; however, it also still compares numbers. When we give any operator, arithmetic, and some functions (toString operator), the ability to do their processing and give us a result for more than its usual value output, this is called polymorphism.

8 Professional practice

8.1 Developing professional work habits

8.1.1 Effective time management and work quality

This class has been one of the most challenging and time-consuming course I have taken at Denison. The sheer amount of workload is immense: exams, reading notes, weekly programming projects, and final portfolio. On average, I spent about 13-15 hours a week to work on class materials and understand key algorithm and data structure concepts. Earlier in the semester, I didn't do well in the first few programming projects. However, my work quality has significantly improved since the first month and I've been doing extremely well in the recent projects. The course also teaches me how to prioritize time and effort. I consistently devote a minimum of 1.5-2 hours every weekday night for this class. In addition to that, I also spend time researching online materials and watching tutorials to go over things that I do not understand.

8.1.2 Working effectively in a small team

I have consistently been working with Kevin in a few projects throughout the semester. In the last few weeks, both of us have been quite busy with different things as second-semester seniors, so we couldn't collaborate in the last 2 projects. However, I have learned a lot from working effectively in groups with him. We figured out a way to split the workload equally and did our individual coding part before merging things together. There were a lot of times that we helped with each other clarify implementation mistakes or fix running time inefficiency. As the old saying goes, "Two brains are better than one."

8.1.3 Class contribution and involvement and being a good department citizen

I personally was very excited about taking CS 271 this semester because I understand that algorithm and data structure are fundamentally important during technical interviews for engineering jobs. Thus, I made sure to be an active contributor to class environment by asking questions, bringing up productive discussion, and facilitating conversations with peers about class concepts and projects. I would always be prepared for class by doing the reading notes and watching tutorial videos beforehand. I always took detailed notes during each class session and if I missed one, I would borrow notes from other classmates.

I enjoyed my time at Denison studying Computer Science. I did strive to go for a Major; however, because of schedule conflict and studying abroad plans, I could only go for a Minor. Nonetheless, I learned valuable things in each of the Computer Science classes that I took, and more importantly, I am now driven to learn things on my own. The Math & CS department has been a familiar place for me throughout my time on the hill, and I will surely miss Olin 2nd floor after graduation.

8.2 Developing effective communication skills

8.2.1 Commenting on code

I read a wonderful book called Clean Code by Robert Martin, which is a must read for any developer, software engineer, project manager, team lead, or system analyst with an interest in producing better code. The book has a whole chapter on comments, when they're good and when they're not. Also a lot of practical advice on how to write code that doesn't need so many comments. Here are good reasons to use comments from the book:

- Legal notices, like copyrights
- When you can't use a function name to explain something
- Your intent behind a decision
- Clarification of code you can't alter, like library call results
- Warning of consequences
- To a reasonable degree
- Amplify the importance of something seemingly inconsequential
- Java Docs in public APIs

8.2.2 Submission and email communication

Submission via NoteBowl has been very simple throughout the course. I took a lot of care in submitting all the programming projects this semester with all the suitable accompanying files and pdf documents. There were a few occasions when I submitted my work late via email due to external circumstances, but I always maintained good communication with the professor to explain the reason.

8.2.3 Learning and using the LaTeX-typesetting system

I have been using LaTeX before in my Proofs Techniques class, so LaTeX is nothing strange with me. I used Overleaf to write LaTeX documents for the majority of the projects, and have grown to be very efficient at that. Writing this portfolio using LaTeX is quite a challenge, but I am glad I was able to complete it.